

Hive and Shark

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)



Motivation

- ▶ MapReduce is **hard** to program.
- ▶ No schema, **lack** of **query languages**, e.g., **SQL**.

- ▶ Adding **tables**, **columns**, **partitions**, and a subset of **SQL** to **unstructured** data.

- ▶ A system for **managing** and **querying structured data** built on top of **Hadoop**.



- ▶ A system for **managing** and **querying structured data** built on top of **Hadoop**.
- ▶ Converts a query to a **series of MapReduce phases**.



- ▶ A system for **managing** and **querying structured data** built on top of **Hadoop**.
- ▶ Converts a query to a **series of MapReduce phases**.
- ▶ Initially developed by **Facebook**.



- ▶ A system for **managing** and **querying structured data** built on top of **Hadoop**.
- ▶ Converts a query to a **series of MapReduce phases**.
- ▶ Initially developed by **Facebook**.
- ▶ Focuses on **scalability** and **extensibility**.



Scalability

- ▶ Massive **scale out** and **fault tolerance** capabilities on **commodity** hardware.
- ▶ Can handle **petabytes** of data.



- ▶ **Data types**: primitive types and complex types.
- ▶ User Defined Functions (**UDF**).
- ▶ **Serializer/Deserializer**: text, binary, JSON ...
- ▶ **Storage**: HDFS, Hbase, S3 ...



RDBMS vs. Hive

	RDBMS	Hive
Language	SQL	HiveQL
Update Capabilities	INSERT, UPDATE, and DELETE	INSERT OVERWRITE; no UPDATE or DELETE
OLAP	Yes	Yes
OLTP	Yes	No
Latency	Sub-second	Minutes or more
Indexes	Any number of indexes	No indexes, data is always scanned (in parallel)
Data size	TBs	PBs



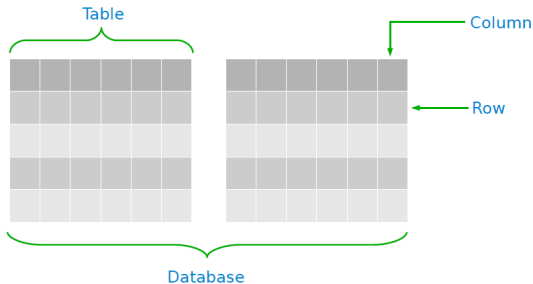
RDBMS vs. Hive

	RDBMS	Hive
Language	SQL	HiveQL
Update Capabilities	INSERT, UPDATE, and DELETE	INSERT OVERWRITE; no UPDATE or DELETE
OLAP	Yes	Yes
OLTP	Yes	No
Latency	Sub-second	Minutes or more
Indexes	Any number of indexes	No indexes, data is always scanned (in parallel)
Data size	TBs	PBs

- ▶ **Online Analytical Processing (OLAP)**: allows users to analyze database information from multiple database systems at one time.
- ▶ **Online Transaction Processing (OLTP)**: facilitates and manages transaction-oriented applications.

Hive Data Model

- ▶ Re-used from **RDBMS**:
 - **Database**: Set of Tables.
 - **Table**: Set of Rows that have the same **schema** (same **columns**).
 - **Row**: A single record; a set of columns.
 - **Column**: provides value and type for a single value.



Hive Data Model - Table

- ▶ Analogous to tables in **relational databases**.
- ▶ Each **table** has a corresponding **HDFS directory**.
- ▶ For example data for table **customer** is in the directory **/db/customer**.

Hive Data Model - Partition

- ▶ A **coarse-grained partitioning** of a table based on the **value** of a **column**, such as a date.
- ▶ **Faster queries** on slices of the data.
- ▶ If **customer** is partitioned on column **country**, then data with a particular country value **SE**, will be stored in files within the directory `/db/customer/country=SE`.

Hive Data Model - Bucket

- ▶ Data in each partition may in turn be divided into buckets based on the `hash of a column` in the table.
- ▶ For more `efficient queries`.
- ▶ If `customer` country partition is subdivided further into buckets, based on `username` (hashed on username), the data for each bucket will be stored within the directories:
`/db/customer/country=SE/000000_0`
...
`/db/customer/country=SE/000000_5`

Column Data Types

- ▶ Primitive types
 - integers, float, strings, dates and booleans
- ▶ Nestable collections
 - array and map
- ▶ User-defined types
 - Users can also define their own types programmatically

- ▶ HiveQL: [SQL-like](#) query languages

Hive Operations

- ▶ HiveQL: **SQL-like** query languages
- ▶ **DDL** operations (**Data Definition Language**)
 - Create, Alter, Drop

Hive Operations

- ▶ HiveQL: **SQL-like** query languages
- ▶ **DDL** operations (**Data Definition Language**)
 - Create, Alter, Drop
- ▶ **DML** operations (**Data Manipulation Language**)
 - Load and Insert (overwrite)
 - Does **not** support **updating** and **deleting**

Hive Operations

- ▶ HiveQL: **SQL-like** query languages
- ▶ **DDL** operations (**Data Definition Language**)
 - Create, Alter, Drop
- ▶ **DML** operations (**Data Manipulation Language**)
 - Load and Insert (overwrite)
 - Does **not** support **updating** and **deleting**
- ▶ **Query** operations
 - Select, Filter, Join, Groupby

DDL Operations (1/3)

► Create tables

```
-- Creates a table with three columns  
CREATE TABLE customer (id INT, name STRING, address STRING)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

DDL Operations (1/3)

▶ Create tables

```
-- Creates a table with three columns  
CREATE TABLE customer (id INT, name STRING, address STRING)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

▶ Create tables with partitions

```
-- Creates a table with three columns and a partition column  
-- /db/customer2/country=SE;  
-- /db/customer2/country=IR;  
CREATE TABLE customer2 (id INT, name STRING, address STRING)  
PARTITION BY (country STRING)
```

DDL Operations (2/3)

► Create tables with buckets

```
-- Specify the columns to bucket on and the number of buckets
-- /db/customer3/000000_0
-- /db/customer3/000000_1
-- /db/customer3/000000_2
set hive.enforce.bucketing = true;
CREATE TABLE customer3 (id INT, name STRING, address STRING)
CLUSTERED BY (id) INTO 3 BUCKETS;
```

DDL Operations (2/3)

► Create tables with buckets

```
-- Specify the columns to bucket on and the number of buckets
-- /db/customer3/000000_0
-- /db/customer3/000000_1
-- /db/customer3/000000_2
set hive.enforce.bucketing = true;
CREATE TABLE customer3 (id INT, name STRING, address STRING)
CLUSTERED BY (id) INTO 3 BUCKETS;
```

► Browsing through tables

```
-- lists all the tables
SHOW TABLES;

-- shows the list of columns
DESCRIBE customer;
```


► Altering tables

```
-- rename the customer table to alaki  
ALTER TABLE customer RENAME TO alaki;  
  
-- add two new columns to the customer table  
ALTER TABLE customer ADD COLUMNS (job STRING);  
ALTER TABLE customer ADD COLUMNS (grade INT COMMENT 'some comment');
```

▶ Altering tables

```
-- rename the customer table to alaki  
ALTER TABLE customer RENAME TO alaki;  
  
-- add two new columns to the customer table  
ALTER TABLE customer ADD COLUMNS (job STRING);  
ALTER TABLE customer ADD COLUMNS (grade INT COMMENT 'some comment');
```

▶ Dropping tables

```
DROP TABLE customer;
```

DML Operations

- ▶ Loading data from flat files.

```
-- if 'LOCAL' is omitted then it looks for the file in HDFS.  
-- the 'OVERWRITE' signifies that existing data in the table is deleted.  
-- if the 'OVERWRITE' is omitted, data are appended to existing data sets.  
LOAD DATA LOCAL INPATH 'data.txt' OVERWRITE INTO TABLE customer;  
  
-- loads data into different partitions  
LOAD DATA LOCAL INPATH 'data1.txt' OVERWRITE INTO TABLE customer2  
PARTITION (country='SE');  
LOAD DATA LOCAL INPATH 'data2.txt' OVERWRITE INTO TABLE customer2  
PARTITION (country='IR');
```

DML Operations

- ▶ Loading data from flat files.

```
-- if 'LOCAL' is omitted then it looks for the file in HDFS.  
-- the 'OVERWRITE' signifies that existing data in the table is deleted.  
-- if the 'OVERWRITE' is omitted, data are appended to existing data sets.  
LOAD DATA LOCAL INPATH 'data.txt' OVERWRITE INTO TABLE customer;  
  
-- loads data into different partitions  
LOAD DATA LOCAL INPATH 'data1.txt' OVERWRITE INTO TABLE customer2  
PARTITION (country='SE');  
LOAD DATA LOCAL INPATH 'data2.txt' OVERWRITE INTO TABLE customer2  
PARTITION (country='IR');
```

- ▶ Store the query results in tables

```
INSERT OVERWRITE TABLE customer SELECT * From old_customers;
```

Query Operations (1/3)

► Selects and filters

```
SELECT id FROM customer2 WHERE country='SE';
```

```
-- selects all rows from customer table into a local directory
```

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/hive-sample-out' SELECT *  
FROM customer;
```

```
-- selects all rows from customer2 table into a directory in hdfs
```

```
INSERT OVERWRITE DIRECTORY '/tmp/hdfs_ir' SELECT * FROM customer2  
WHERE country='IR';
```

Query Operations (2/3)

► Aggregations and groups

```
SELECT MAX(id) FROM customer;
```

```
SELECT country, COUNT(*), SUM(id) FROM customer2 GROUP BY country;
```

```
INSERT TABLE high_id_customer SELECT c.name, COUNT(*) FROM customer c  
WHERE c.id > 10 GROUP BY c.name;
```

Query Operations (3/3)

▶ Join

```
CREATE TABLE customer (id INT, name STRING, address STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

CREATE TABLE order (id INT, cus_id INT, prod_id INT, price INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

SELECT * FROM customer c JOIN order o ON (c.id = o.cus_id);

SELECT c.id, c.name, c.address, ce.exp FROM customer c JOIN
(SELECT cus_id, sum(price) AS exp FROM order GROUP BY cus_id) ce
ON (c.id = ce.cus_id) INSERT OVERWRITE TABLE order_customer;
```

User-Defined Function (UDF)

```
package com.example.hive.udf;  
  
import org.apache.hadoop.hive.ql.exec.UDF;  
import org.apache.hadoop.io.Text;  
  
public final class Lower extends UDF {  
    public Text evaluate(final Text s) {  
        if (s == null) { return null; }  
        return new Text(s.toString().toLowerCase());  
    }  
}
```

```
-- Register the class  
CREATE FUNCTION my_lower AS 'com.example.hive.udf.Lower';  
  
-- Using the function  
SELECT my_lower(title), sum(freq) FROM titles GROUP BY my_lower(title);
```


Executing SQL Questions

- ▶ Processes **HiveQL statements** and generates the **execution plan** through three-phase processes.
 - ① **Query parsing**: transforms a query string to a parse tree representation.
 - ② **Logical plan generation**: converts the internal query representation to a logical plan, and **optimizes** it.
 - ③ **Physical plan generation**: split the optimized logical plan into multiple map/reduce and HDFS tasks.

Optimization (1/2)

- ▶ Column pruning
 - Projecting out the **needed columns**.
- ▶ Predicate pushdown
 - Filtering rows **early** in the processing, by pushing down predicates to the scan (if possible).
- ▶ Partition pruning
 - Pruning out files of partitions that do not satisfy the predicate.

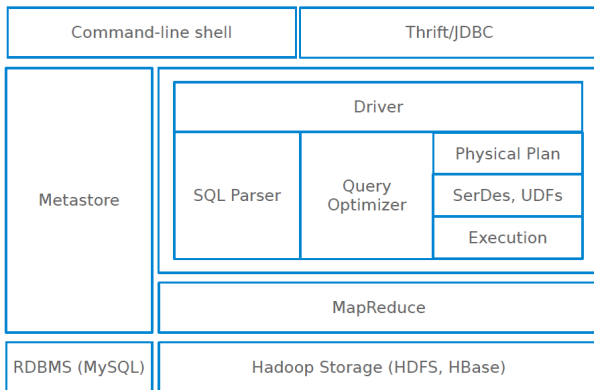
▶ Map-side joins

- The **small tables** are **replicated** in all the mappers and joined with other tables.
- **No reducer** needed.

▶ Join reordering

- Only materialized and kept **small tables** in **memory**.
- This ensures that the join operation does not exceed memory limits on the reducer side.

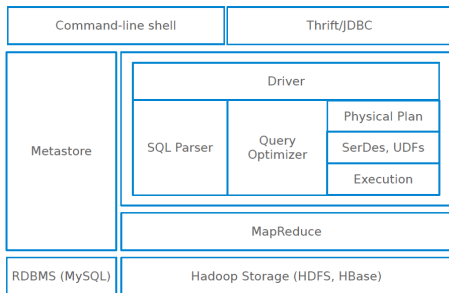
Hive Components (1/8)



Hive Components (2/8)

▶ External interfaces

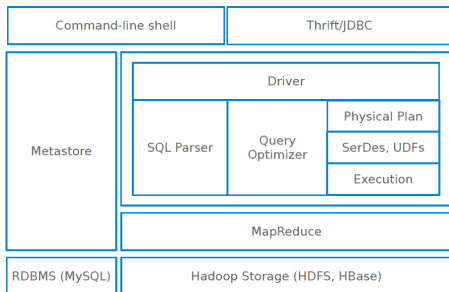
- User interfaces, e.g., CLI and web UI
- Application programming interfaces, e.g., JDBC and ODBC
- [Thrift](#), a framework for **cross-language services**.



Hive Components (3/8)

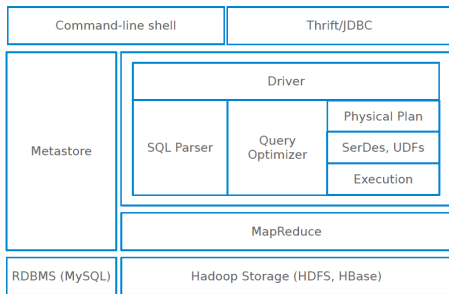
► Driver

- Manages the **life cycle** of a HiveQL statement during compilation, optimization and execution.



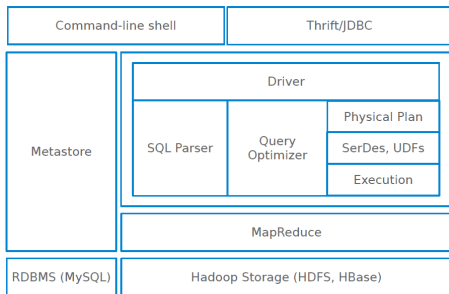
Hive Components (4/8)

- ▶ Compiler (Parser/Query Optimizer)
 - Translates the HiveQL statement into a logical plan, and optimizes it.



Hive Components (5/8)

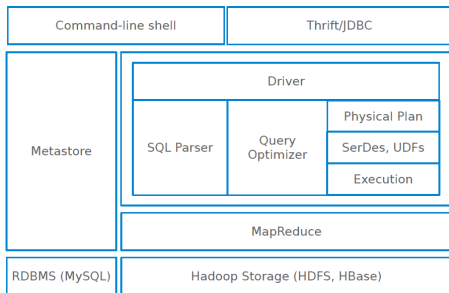
- ▶ Physical plan
 - Transforms the logical plan into a **DAG of Map/Reduce jobs**.



Hive Components (6/8)

▶ Execution engine

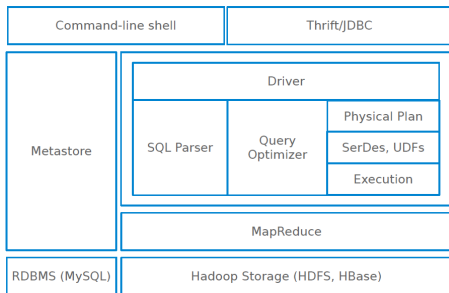
- The driver submits the individual mapreduce jobs from the DAG to the execution engine in a **topological order**.



Hive Components (7/8)

► SerDe

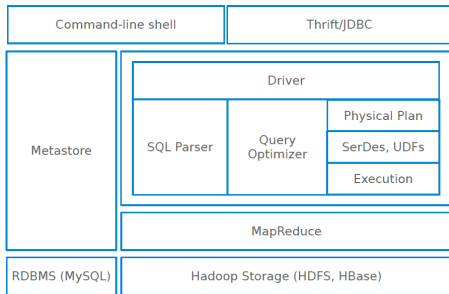
- **Serializer/Deserializer** allows Hive to read and write table rows in any **custom format**.



Hive Components (8/8)

► Metastore

- The **system catalog**.
- Contains **metadata** about the tables.
- Metadata is **specified** during table **creation** and **reused** every time the table is referenced in HiveQL.
- Metadatas are stored on either a traditional **relational database**, e.g., MySQL, or **file system** and **not HDFS**.



Hive on Spark

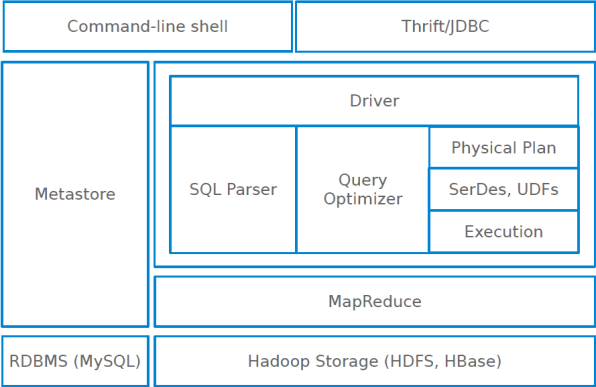


- ▶ **RDDs** are **immutable**, partitioned **collections** that can be created through various **transformations**, e.g., map, groupByKey, join.

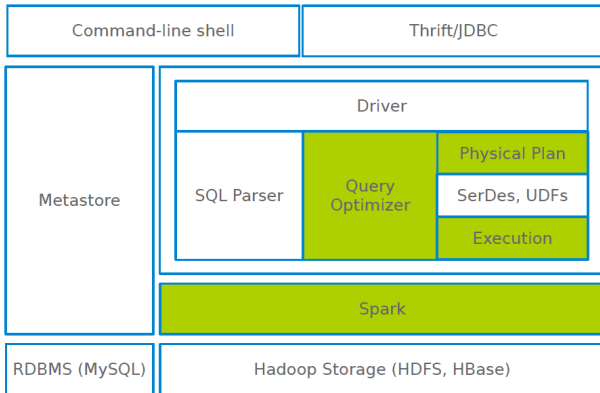
Executing SQL over Spark RDDs

- ▶ Shark runs SQL queries over Spark using **three-step process**:
 - ① **Query parsing**: Shark uses Hive query compiler to parse the query and generate a parse tree.
 - ② **Logical plan generation**: the tree is turned into a logical plan and basic logical optimization is applied.
 - ③ **Physical plan generation**: Shark applies additional optimization and creates a physical plan consisting of transformations on RDDs.

Hive Components



Shark Components



- ▶ Shark **extended RDD execution model**:
 - **Partial DAG Execution (PDE)**: to re-optimize a running query after running the first few stages of its task DAG.
 - **In-memory columnar storage** and compression: to process relational data efficiently.
 - Control over **data partitioning**.

Partial DAG Execution (1/2)

- ▶ How to optimize the following query?

```
SELECT * FROM table1 a JOIN table2 b ON (a.key = b.key)
WHERE my_crazy_udf(b.field1, b.field2) = true;
```

Partial DAG Execution (1/2)

- ▶ How to optimize the following query?

```
SELECT * FROM table1 a JOIN table2 b ON (a.key = b.key)
WHERE my_crazy_udf(b.field1, b.field2) = true;
```

- ▶ It can not use **cost-based optimization** techniques that rely on accurate **a priori data statistics**.
- ▶ They require dynamic approaches to query optimization.
- ▶ **Partial DAG Execution (PDE)**: **dynamic alteration** of query plans based on data statistics collected at **run-time**.

Partial DAG Execution (2/2)

- ▶ The workers gather customizable statistics at **global** and **per-partition** granularities at run-time.
- ▶ Each **worker** sends the collected statistics to the **master**.
- ▶ The **master** aggregates the statistics and **alters** the **query plan** based on such statistics.

Columnar Memory Store

- ▶ Simply caching Hive records as **JVM objects** is **inefficient**.
- ▶ 12 to 16 bytes of overhead per object in JVM implementation:
 - e.g., storing a 270MB table as JVM objects uses approximately 971 MB of memory.
- ▶ Shark employs **column-oriented** storage using arrays of primitive objects.

1	John	4.1
2	mike	3.5
3	sally	6.4

Row Storage

1	2	3
john	mike	sally
4.1	3.5	6.4

Column Storage

- ▶ Shark allows **co-partitioning** two tables, which are **frequently joined together**, on a common key for faster joins in subsequent queries.

Shark/Spark Integration

- ▶ Shark provides a simple API for programmers to convert results from SQL queries into a special type of RDDs: `sql2rdd`.

```
val youngUsers = sql2rdd("SELECT * FROM users WHERE age < 20")  
  
println(youngUsers.count)  
  
val featureMatrix = youngUsers.map(extractFeatures(_))  
  
kmeans(featureMatrix)
```

- ▶ Operators: DDL, DML, SQL
- ▶ Hive architecture vs. Shark architecture
- ▶ Add advance features to Spark, e.g., PDE, columnar memory store

Questions?